CIS 552: Advanced Programming

- Home
- Schedule
- Resources
- Style guide
- Syllabus

Quickcheck and Monad Transformers

Preliminaries

To complete this homework, you will need to download and edit Main.hs and Sat.hs as plain text. You will also need WhilePP.lhs, although you will not need to modify this file. Your code must typecheck against the given type signatures. Remember to add your own tests to this file to exercise the functions you write. As always, submit your homework by uploading to the course website.

Problem 0: A SAT Solver

See Sat.html.

Problem 1: An Interpreter for WHILE++

```
{-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults -fno-warn-orphans #-}
{-# LANGUAGE TypeSynonymInstances, FlexibleContexts, NoMonomorphismRestriction, FlexibleInsta

module Main where

import WhilePP

import Data.Map (Map)
import qualified Data.Map as Map

import Control.Monad.State
import Control.Monad.Error
import Control.Monad.Writer

import Test.HUnit hiding (State)
```

Previously, you wrote a simple interpreter for *WHILE*. For this problem, you will use monad transformers to build an evaluator for *WHILE++* which, adds exceptions and I/O to the original language.

The only new constructs are the `Print`, `Throw` and the `Try` statements.

- `Print s e` should print out (eg to stdout) the string corresponding to the string `s` followed by whatever `e` evaluates to, followed by a newline --- for example,`Print "Three: " (IntVal 3)' should display "Three: IntVal 3",

- `Throw e` evaluates the expression `e` and throws it as an exception, and

- `Try s x h` executes the statement `s` and if in the course of execution, an exception is thrown, then the exception comes shooting up and is assigned to the variable `x` after which the *handler* statement `h` is executed.

We will use the `State` monad to represent the world-transformer. Intuitively, `State s a` is equivalent to the world-transformer `s -> (a, s)`. See the above documentation for more details. You can ignore the bits about `StateT` for now.

Use monad transformers to write a function

```
type Store    = Map Variable Value
evalS :: (MonadState Store m, MonadError Value m, MonadWriter String m) => Statement -> m ()
evalS = undefined
```

and use the above function to implement a second function

```
execute :: Store -> Statement -> (Store, Maybe Value, String)
execute = undefined
```

such that `execute st s` returns a triple `(st', exn, log)` where

- `st'` is the output state,
- `exn` is possibly an exception (if the program terminates with an uncaught exception),
- `log` is the log of messages generated by the `Print` statements.

Requirements

In the case of exceptional termination, the `st'` should be the state *at the point where the last exception was thrown, and `log` should include all the messages upto that point -- make sure you stack your transformers appropriately!

- Reading an undefined variable should raise an exception carrying the value `IntVal 0`.

```
raises :: Statement -> Value -> Test
s `raises` v = case (execute Map.empty s) of
    (_, Just v', _) -> v ~?= v'
    _  -> undefined

t1 :: Test
t1 = (Assign "X"  (Var "Y")) `raises` IntVal 0
```

- Division by zero should raise an exception carrying the value `IntVal 1`.

```
t2 :: Test
t2 = (Assign "X" (Op Divide (Val (IntVal 1)) (Val (IntVal 0)))) `raises` IntVal 1
```

- A run-time type error (addition of an integer to a boolean, comparison of non-integer values) should raise an exception carrying the value `IntVal 2`.

```
t3 :: Test
t3 = TestList [ Assign "X" (Op Plus (Val (IntVal 1)) (Val (BoolVal True))) `raises` IntVal 2,
                If (Val (IntVal 1)) Skip Skip `raises` IntVal 2,
                While (Val (IntVal 1)) Skip `raises` IntVal 2]
```

Example 1

If `st` is the empty state (all variables undefined) and `s` is the program

```
X := 0 ;
Y := 1 ;
print "hello world: " X;
if X < Y then
  throw (X+Y)
else
  skip
endif;
Z := 3
```

represented in Haskell as:

```
mksequence :: [Statement] -> Statement
mksequence = foldr Sequence Skip

testprog1 :: Statement
testprog1 = mksequence [Assign "X" $ Val $ IntVal 0,
                        Assign "Y" $ Val $ IntVal 1,
                        Print "hello world: " $ Var "X",
                        If (Op Lt (Var "X") (Var "Y")) (Throw (Op Plus (Var "X") (Var "Y")))
                                                       Skip,
                        Assign "Z" $ Val $ IntVal 3]
```

then the following test should pass:

```
t4 :: Test
t4 = execute Map.empty testprog1 ~?=
  (Map.fromList [("X", IntVal 0), ("Y",  IntVal 1)], Just (IntVal 1), "hello world: 0")
```

Example 2

If `st` is the empty state (all variables undefined) and `s` is the program

```
X := 0 ;
Y := 1 ;
try
  if X < Y then
    A := 100;
    throw (X+Y);
    B := 200;
  else
    skip
  endif;
catch E with
  Z := E + A
endwith
```

represented in Haskell as

```
testprog2 :: Statement
testprog2 = mksequence [Assign "X" $ Val $ IntVal 0,
                        Assign "Y" $ Val $ IntVal 1,
                        Try (If (Op Lt (Var "X") (Var "Y"))
                                (mksequence [Assign "A" $ Val $ IntVal 100,
                                             Throw (Op Plus (Var "X") (Var "Y")),
                                             Assign "B" $ Val $ IntVal 200])
                                Skip)
                            "E"
                            (Assign "Z" $ Op Plus (Var "E") (Var "A"))]
```

then the following test should pass:

```
t5 :: Test
t5 = execute Map.empty testprog2 ~?=
   ( Map.fromList [("A", IntVal 100), ("E", IntVal 1)
          ,("X", IntVal 0), ("Y", IntVal 1)
      ,("Z", IntVal 101)]
          , Nothing
   , "")

main :: IO ()
main = do
   _ <- runTestTT $ TestList [ t1, t2, t3, t4, t5 ]
   return ()
```

News :

Welcome to CIS 552!
See the home page for basic information about the course, the schedule for the lecture notes and assignments, the resources for links to the required software and online references, and the syllabus for detailed information about the course policies.

Links :

- Piazza
- Haskell.org
- GHC manual
- Library documentation
- Hackage

Design by Minimalistic Design
Powered by Pandoc