My Report

Me

Thursday 13th March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 How to use this?

To generate the PDF, open `report.tex` in your favorite LaTeXeditor and compile. Alternatively, you can manually do `pdflatex report; bibtex report; pdflatex report; pdflatex report` in a terminal.

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.
- To open ghci and play with your code: `stack ghci`
- To run the executable from Section 4: `stack build && stack exec myprogram`
- To run the tests from Section 5: `stack clean && stack test --coverage`

2 The most basic library

This section describes a module which we will import later on.

```haskell
module Basics where

import Control.Monad
import System.Random

thenumbers :: [Integer]
thenumbers = [1..]

somenumbers :: [Integer]
somenumbers = take 10 thenumbers

randomnumbers :: IO [Integer]
randomnumbers = replicateM 10 $ randomRIO (0,10)
```

We can interrupt the code anywhere we want.

```haskell
funnyfunction :: Integer -> Integer
funnyfunction 0 = 42
```

Even in between cases, like here. It's always good to cite something [Knu11].

```haskell
funnyfunction n | even n    = funnyfunction (n-1)
                | otherwise = n*100
```

Something to reverse lists.

```haskell
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = myreverse xs ++ [x]
```

If you look at the `.lhs` file then below this line you can find some Haskell code.

But it does not show up in the PDF document. Please only use this for boring or repetitive parts of your code. Do not hide too much from your reader.

That's it, for now.

3 Implementing the KL-semantics

```haskell
module SemanticsKL where

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

We represent standard names as strings (e.g., "n1", "n2", ...) and have (in theory) an infinite amount of them. We also define a newtype Variable, which we also represent as strings.

```haskell
newtype StdName = StdName String deriving (Eq, Ord, Show)

newtype Variable = Var String deriving (Eq, Ord, Show)
```

Terms are variables, standard names, or function applications, Atomic propositions are predicates applied to terms.

```haskell
data Term = VarTerm Variable
          | StdNameTerm StdName
          | FuncApp String [Term]
          deriving (Eq, Ord, Show)

data Atom = Pred String [Term] deriving (Eq, Ord, Show)
```

Based on this, we can now define $\mathcal{KL}$-formulas:

```haskell
data Formula = Atom Atom                -- Predicate (e.g. Teach(x, "n1"))
              | Equal Term Term          --Equality (e.g., x = "n1")
              | Not Formula              -- Negation
              | Or Formula Formula       -- Disjunction
              | Exists Variable Formula -- Existential (e.g., exists x (Teach x "sue"))
                 TODO
              | K Formula                -- Knowledge Operator (e.g., K (Teach "ted" "sue"))
              deriving (Show)

-- TODO: model forall, rightarrow, leftrightarrow
```

We want a world state to assign values to primitive terms and atoms, so we create a new type WorldState. An epistemic state is a set of possible world states.

```haskell
data WorldState = WorldState
  { atomValues :: Map Atom Bool        -- Truth values of ground atoms
  , termValues :: Map Term StdName     -- Values of ground terms
  } deriving (Eq, Ord, Show)

type EpistemicState = Set WorldState
```

To evaluate a ground term in a world state, we define a function evalTerm that takes a WorldState and a Term and returns a StdName. The idea is to map syntactic terms to their semantic values (standard names) in a given world state. The function uses pattern matching to handle the three possible forms of Term:

1. VarTerm _
   If the term is a variable (e.g., x), it throws an error. This enforces a precondition that evalTerm only works on ground terms (terms with no free variables). In $\mathcal{KL}$, variables must be substituted with standard names before evaluation, aligning with the semantics where only ground terms have denotations This is a runtime check to catch ungrounded inputs.

2. StdNameTerm n
   If the term is a standard name wrapped in StdNameTerm (e.g., StdNameTerm (StdName "n1")), it simply returns the underlying StdName (e.g., StdName "n1"). Standard names in $\mathcal{KL}$ are constants that denote themselves (ibid., p.22). For example, if n=StdName "n1", it represents the individual n1, and its value in any world is n1. In this case, no lookup or computation is needed.

3. FuncApp f args
   If the term is a function application (e.g., f(n1,n2)), evalTerm evaluates the argument, by recursively computing the StdName values of each argument in args using evalTerm w. Next, the ground term is constructed: It Builds a new FuncApp term where all arguments are standard names (wrapped in StdNameTerm), ensuring it's fully ground. We then look up the value by querying the termValues map in the world state w for the denotation of this ground term, defaulting to StdName "n1" if not found.

```
evalTerm :: WorldState -> Term -> StdName
evalTerm w t = case t of
  VarTerm _ -> error "evalTerm: Variables must be substituted"
  StdNameTerm n -> n
  FuncApp f args ->
    let argValues = map (evalTerm w) args
        groundTerm = FuncApp f (map StdNameTerm argValues)
    in Map.findWithDefault (StdName "n1") groundTerm (termValues w) -- Default for
        undefined, use "n1"
```

Next, we want to define a satisfies function. For this, we need a helper-function that checks whether a term or formula is ground. We also need a helper-function that substitutes a variable with a standard name in a term for the exists-case.

```
-- Check if a term is ground (contains no variables).
isGround :: Term -> Bool
isGround t = case t of
  VarTerm _ -> False
  StdNameTerm _ -> True
  FuncApp _ args -> all isGround args

-- Check if a formula is ground.
isGroundFormula :: Formula -> Bool
isGroundFormula f = case f of
  Atom (Pred _ terms) -> all isGround terms
  Equal t1 t2 -> isGround t1 && isGround t2
  Not f' -> isGroundFormula f'
  Or f1 f2 -> isGroundFormula f1 && isGroundFormula f2
  Exists _ _ -> False   -- always contains a variable
  K f' -> isGroundFormula f'

-- Substitute a variable with a standard name in a term.
substTerm :: Variable -> StdName -> Term -> Term
substTerm x n t = case t of
  VarTerm v | v == x -> StdNameTerm n
  VarTerm _ -> t
  StdNameTerm _ -> t
  FuncApp f args -> FuncApp f (map (substTerm x n) args)

-- Substitute a variable with a standard name in a formula.
subst :: Variable -> StdName -> Formula -> Formula
subst x n formula = case formula of
  Atom (Pred p terms) -> Atom (Pred p (map (substTerm x n) terms))
  Equal t1 t2 -> Equal (substTerm x n t1) (substTerm x n t2)
  Not f -> Not (subst x n f)
  Or f1 f2 -> Or (subst x n f1) (subst x n f2)
  Exists y f | y == x -> formula -- Avoid capture
             | otherwise -> Exists y (subst x n f)
  K f -> K (subst x n f)
```

Since we want to check for satisfiability in a model, we want to make the model explicit:

```
data Model = Model
  { actualWorld :: WorldState       -- The actual world state
  , epistemicState :: EpistemicState -- Set of possible world states
  , domain :: Set StdName           -- Domain of standard names
  } deriving (Show)
```

With the helper-functions and the new type Model, we can now implement the satisfiesModel function as follows:

```
satisfiesModel :: Model -> Formula -> Bool
satisfiesModel m = satisfies (epistemicState m) (actualWorld m)
  where
    satisfies e w formula = case formula of
      Atom (Pred p terms) ->
        if all isGround terms
          then Map.findWithDefault False (Pred p terms) (atomValues w)
          else error "Non-ground atom in satisfies!"
      Equal t1 t2 ->
        if isGround t1 && isGround t2
          then evalTerm w t1 == evalTerm w t2
          else error "Non-ground equality in satisfies!"
      Not f ->
        not (satisfies e w f)
      Or f1 f2 ->
        satisfies e w f1 || satisfies e w f2
      Exists x f ->
      -- \(e, w \models \exists x. \alpha\) iff for some name \(n\), \(e, w \models \
          alpha_n^x\)
        any (\n -> satisfies e w (subst x n f)) (Set.toList $ domain m)
      -- \(e, w \models K \alpha\) iff for every \(w' \in e\), \(e, w' \models \alpha\)
      K f ->
        all (\w' -> satisfies e w' f) e
```

Building on this we can implement a function checkModel that checks whether a formula holds in a given model:

```
checkModel :: Model -> Formula -> Bool
checkModel m phi = all (satisfiesModel m) (groundFormula phi (domain m))
```

Note that we use the function groundFormula here. Since we have implemented satisfiesModel such that it assumes ground formulas or errors out, we decided to handle free variables by grounding formulas by substituting free variables. We implement groundFormula as follows:

```
groundFormula :: Formula -> Set StdName -> [Formula]
groundFormula f domain = do
  -- converts the set of free variables in f to a list
  let fvs = Set.toList (freeVars f)
  -- creates a list of all possible assignments of domain elements to each free variable
  -- For each variable in fvs, toList domain provides the list of standard names, mapM
      applies this (monadically), producing all the combinations
  subs <- mapM (\_ -> Set.toList domain) fvs
  --iteratively substitute each variable v with a standard name n in the formula
  return $ foldl (\acc (v, n) -> subst v n acc) f (zip fvs subs)
```

This function takes a formula and a domain of standard names and returns a list of all possible ground instances of the formula by substituting its free variables with elements from the domain. We use a function freeVars that identifies all the variables in a formula that need grounding or substitution. It takes a formula and returns a Set Variable containing all the free variables in that formula:

```
freeVars :: Formula -> Set Variable
freeVars f = case f of
  --Collects free variables from all terms in the predicate
  Atom (Pred _ terms) -> Set.unions (map freeVarsTerm terms)
  -- Unions the free variables from both terms t1 and t2.
  Equal t1 t2 -> freeVarsTerm t1 ‘Set.union‘ freeVarsTerm t2
  -- Recursively computes free variables in the negated subformula f’.
  Not f’ -> freeVars f’
  -- Unions the free variables from both disjuncts f1 and f2.
  Or f1 f2 -> freeVars f1 ‘Set.union‘ freeVars f2
  -- Computes free variables in f’, then removes x (the bound variable) using delete, since
       x is not free within \exists x f’
  Exists x f’ -> Set.delete x (freeVars f’)
  -- Recursively computes free variables in f’, as the K operator doesn’t bind variables.
  K f’ -> freeVars f’
  where
    freeVarsTerm t = case t of
      --A variable (e.g., x) leads to a singleton set containing v.
      VarTerm v -> Set.singleton v
      -- A standard name (e.g., n1) has no free variables, so returns an empty set.
      StdNameTerm _ -> Set.empty
      -- A function application (e.g., f(x,n1)) recursively computes free variables in its
          arguments.
      FuncApp _ args -> Set.unions (map freeVarsTerm args)
```

4 Wrapping it up in an exectuable

We will now use the library form Section 2 in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
  print somenumbers
  print (map funnyfunction somenumbers)
  myrandomnumbers <- randomnumbers
  print myrandomnumbers
  print (map funnyfunction myrandomnumbers)
  putStrLn "GoodBye"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

5 Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```haskell
module Main where

import Basics

import Test.Hspec
import Test.QuickCheck
```

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

```haskell
main :: IO ()
main = hspec $ do
  describe "Basics" $ do
    it "somenumbers should be the same as [1..10]" $
      somenumbers `shouldBe` [1..10]
    it "if n > - then funnyfunction n > 0" $
      property (\n -> n > 0 ==> funnyfunction n > 0)
    it "myreverse: using it twice gives back the same list" $
      property $ \str -> myreverse (myreverse str) == (str::String)
```

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test` Then look for "The coverage report for ... is available at ... .html" and open this file in your browser. See also: `https://wiki.haskell.org/Haskell_program_coverage`.

6 Conclusion

Finally, we can see that [LW13] is a nice paper.

References

[Knu11] Donald E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms, Part 1*, volume 4A. Addison-Wesley Professional, 2011.

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.