Matt Parsons
11-4-2015
CSCI 4550

1 DPLL, Australia, and You

Let's do an exploratory coding session with the DPLL algorithm! While it's normally a good idea to sit down, plan your data types, and try to make everything make sense at first, sometimes it is more fun to just write garbage code and get it to work.

This is a literate Haskell document.

The `main` function defined later can be run directly, or the functions defined here can be opened in a REPL.

```
$ stack build          -- installs necessary packages
$ stack runghc hw6.lhs -- prints Australia
$ stack ghci hw6.lhs   -- opens REPL
```

As usual, we'll start with some imports.

```
module HW6 where

import           Control.Monad    (guard, when)
import           Control.Monad.ST (ST (), runST)
import           Data.Foldable    (asum, for_, toList)
import           Data.List        (nub, (\\))
import           Data.Map         (Map ())
import qualified Data.Map         as Map
import           Data.Maybe       (listToMaybe, mapMaybe, fromJust)
import           Data.Set         (Set ())
import qualified Data.Set         as Set
import           Data.STRef
import           Data.Traversable (for)
```

Our assignment is to implement the DPLL algorithm. Let's just write it down and see how it goes.

```
dpll0 :: Clauses -> Symbols -> Model -> Bool
dpll0 clauses symbols model =
  if all (`isTrueIn` model) clauses
  then True
  else
    if any (`isFalseIn` model) clauses
    then False
    else
      case findPureSymbol' symbols clauses model of
```

```
          (Just sym, val) ->
            dpll0 clauses (symbols `minus` sym)
                          (model `including` (sym := val))
          (Nothing, val) ->
            case findUnitClause' clauses model of
                 (Just sym, val) ->
                   dpll0 clauses (symbols `minus` sym)
                                 (model `including` (sym := val))
                 (Nothing, val) ->
                   case symbols of
                        (x:xs) ->
                          dpll0 clauses xs (model `including` (x := False))
                          || dpll0 clauses xs (model `including` (x := True))
 where
   findUnitClause' = undefined
   findPureSymbol' = undefined
```

This is essentially a transcription of the algorithm in the book to Haskell expressions. No care has been made that the names or types actually *mean* anything. In fact, to avoid clashing, we need to keep the two `find` functions undefined... Let's rearrange and refactor the code to make it a bit nicer.

First, we factor the if statements into guard expressions. Then, noticing that the `findPureSymbol` and `findUnitClause` functions both return something of a `Maybe` value, we can potentially make that into somewhat nicer monadic expression. Punting that decision, we get this version:

```
dpll1 :: Clauses -> Symbols -> Model -> Bool
dpll1 clauses symbols model
  | all (`isTrueIn` model) clauses  = True
  | any (`isFalseIn` model) clauses = False
  | otherwise =
      case findPureSymbol symbols clauses model of
           Just (sym := val) ->
             dpll1 clauses (symbols `minus` sym)
                           (model `including` (sym := val))
           Nothing ->
             case findUnitClause clauses model of
                  Just (sym := val) ->
                    dpll1 clauses (symbols `minus` sym)
                                  (model `including` (sym := val))
                  Nothing ->
                    case symbols of
                         (x:xs) ->
                           dpll1 clauses xs (model `including` (x := False))
                           || dpll1 clauses xs (model `including` (x := True))
```

Next, we'll factor the branches out into named subexpressions in a where clause, and use the `maybe` function to handle the branching. `maybe` takes a default value of type `b`, a function from `a` to `b`, and

a value of `Maybe a`. If the `Maybe` value has `Just a`, then it applies the function to `a`. Otherwise, it returns the default value provided.

```
dpll2 :: Clauses -> Symbols -> Model -> Bool
dpll2 clauses symbols@(x:xs) model
  | all (`isTrueIn` model) clauses  = True
  | any (`isFalseIn` model) clauses = False
  | otherwise =
      maybe pureSymbolNothing pureSymbolJust (findPureSymbol symbols clauses model)
  where
    pureSymbolJust (sym := val) =
      dpll2 clauses (symbols `minus` sym) (model `including` (sym := val))
    pureSymbolNothing =
      maybe unitClauseNothing unitClauseJust (findUnitClause clauses model)
    unitClauseJust (sym := val) =
      dpll2 clauses (symbols `minus` sym) (model `including` (sym := val))
    unitClauseNothing =
      dpll2 clauses xs (model `including` (x := False))
      || dpll2 clauses xs (model `including` (x := True))
```

This is still kind of ugly. Can we do better?

We can! The general pattern here is "Try this option. If it fails, try the next." Haskell has a type class called `Alternative` with the following operations:

- `a <|> b`, a binary operator saying "Choose either a or b"
- `empty`, an identity for the above operator.

`Maybe` has an `Alternative` instance, which skips over `Nothing` values. That will let us use `asum`, which will take the first `Just` value from a list. We can use this to build a list of things to do, and take the first successful action.

There's a common pattern in all of the above branches: we call `dpll` with a slightly different set of arguments. In each case, we include something new in the model, and remove that from the available symbols. We'll extract that logic into the `next` function.

Since we also want it to return the model it used, we'll indicate a false result with Nothing, and a True result with Just model.

```
dpll :: Clauses -> Symbols -> Model -> Maybe Model
dpll clauses symbols model
  | all (`isTrueIn` model) clauses  = Just model
  | any (`isFalseIn` model) clauses = Nothing
  | otherwise =
      asum (map (>>= next) controlList)
  where
    next :: Assignment -> Maybe Model
    next (sym := val) =
```

```
      dpll clauses (symbols `minus` sym) (model `including` (sym := val))

controlList :: [Maybe Assignment]
controlList =
  [ findPureSymbol symbols clauses model
  , findUnitClause clauses model
  , (:= False) <$> listToMaybe symbols
  , (:= True) <$> listToMaybe symbols
  ]
```

There we go! This is more like the Haskell we know and love. We take our control list, bind the `next` function to each possible assignment, and use `asum` to take the first item that has a `Just` value. While this might seem terribly inefficient, Haskell's laziness ensures that these operations only do work as the work is needed.

Well, now we've done a rather remarkable amount of work without doing anything at all! Let's start defining some types and functions so the above transcription actually means something. The program definitely isn't compiling at this point: the types `Clauses`, `Symbols`, etc. aren't defined at all. The functions `isTrueIn`, `including`, `minus`, and `:=` aren't defined either. Let's analyze what we're doing with the terms to get some idea of what data types will fit for them. A symbol could be anything – but we'll just use a String for now.

```
type Symbol = String
```

We've used `listToMaybe` on the symbols. This betrays our intent – `Symbols` is going to be a list. That gives us enough information to define `minus` as well.

```
type Symbols = [Symbol]

minus :: Symbols -> Symbol -> Symbols
minus xs = (xs \\) . pure
```

The `Clauses` type is a collection of expressions. An expression is an organization of terms in CNF.

```
type Clauses = [CNF]
```

With CNF, juxtaposion is conjunctions. We can therefore represent a CNF expression as a list of literals.

```
type CNF = [Literal]
```

Finally, a literal is a pair of Sign and Symbol.

```
type Literal = (Sign, Symbol)
```

Of course, a sign is just a function that either negates or doesn't negate a boolean expression.

```haskell
data Sign = Pos | Neg
  deriving (Eq, Ord, Show)

apply :: Sign -> Bool -> Bool
apply Pos = id
apply Neg = not

n :: Symbol -> Literal
n c = (Neg, c)

p :: Symbol -> Literal
p c = (Pos, c)
```

We now have enough information to define the clauses given in the assignment:

```haskell
ex :: Clauses
ex =
  [ [ n "p", p "a", p "c" ]
  , [ n "a" ]
  , [ p "p", n "c" ]
  ]
```

The main use we have for the clauses type is to map over it with `isTrueIn` and `isFalseIn`, checking every clause in the list against the model. The model is an assignment of symbols to truth values, and random access time is important. We'll use a map.

```haskell
type Model = Map Symbol Bool

isTrueIn :: CNF -> Model -> Bool
isTrueIn cnf model = or (mapMaybe f cnf)
  where
    f (sign, symbol) = apply sign <$> Map.lookup symbol model
```

Here, we're looking up every symbol in the CNF expression, and applying the literal's sign to the possible value. If there's no value in the model, then it doesn't return anything. `or` checks to see if there are any `True` values in the resulting list. If there are, then the CNF is true in the model.

```haskell
isFalseIn :: CNF -> Model -> Bool
isFalseIn cnf model = all not literals
  where
    literals =
      map f cnf
    f (sign, symbol) =
      apply sign (Map.findWithDefault (apply sign True) symbol model)
```

`isFalseIn` is trickier – we map over the CNF expression with a default value of the `sign` applied to `True`. Then, we apply the `sign` again to the resulting value. `all not` is a way of saying "every element is false."

Now the compiler is complaining about not recognizing the `:=` symbol. As it happens, any infix function that prefixed with a `:` is a data constructor. We'll define the data type `Assignment` and give it some accessor functions.

```
data Assignment
  = (:=)
  { getSymbol :: Symbol
  , getValue :: Bool
  }

instance Show Assignment where
  show (s := v) = "(" ++ s ++ " := " ++ show v ++ ")"
```

An advantage of using a data constructor is that we can pattern match on the values of that constructor. This gives us a rather nice definition of the `including`:

```
including :: Model -> Assignment -> Model
including m (sym := val) = Map.insert sym val m
```

The final remaining items that aren't defined are findPureSymbol and findUnitClause. From the textbook,

> Pure symbol heuristic: A pure symbol is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure.

If a symbol has all negative signs, then the returned assignment is False. If a symbol has all positive signs, then the returned assignment is True. We'll punt refining the clauses with the model to a future function...

```
findPureSymbol :: Symbols -> Clauses -> Model -> Maybe Assignment
findPureSymbol symbols clauses' model =
  asum (map makeAssignment symbols)
  where
    clauses = refinePure clauses' model
```

We're using `asum` again to pick the first assignment that works out. We'll map the `makeAssignment` function over the list of symbols.

```
    makeAssignment :: Symbol -> Maybe Assignment
    makeAssignment sym =
      (sym :=) <$> negOrPos (signsForSymbol sym)
```

This maps the assignment of the sym variable over the `negOrPos` function, which determines whether the symbol should have a True or False assignment.

```
signsForSymbol :: Symbol -> [Sign]
signsForSymbol sym =
  clauses >>= signsForSymInClause sym
```

`signsForSymbol` binds the clauses, and for each clause, gets the list of signs associated with the symbol. The resulting list of lists is concatenated.

```
signsForSymInClause :: Symbol -> CNF -> [Sign]
signsForSymInClause sym =
  map fst . filter ((== sym) . snd)
```

`fst` extracts the first element from a tuple: `fst (a, _) = a`. `snd` extracts the second element: `snd (_, a) = a`. So we're filtering the `CNF` (a list of `Literal`s) to only contain the elements whose second element is equal to the symbol. And then we're extracting the first element, leaving just the `Sign`s.

```
negOrPos :: [Sign] -> Maybe Bool
negOrPos = getSingleton . nub . applyTrue

applyTrue :: [Sign] -> [Bool]
applyTrue = map (`apply` True)

getSingleton :: [a] -> Maybe a
getSingleton [x] = Just x
getSingleton _   = Nothing
```

`nub` is Haskell's rather confusingly named "remove duplicate elements" function. In any case, we can't test functions for equality, so we have to apply a value to the `Sign`s to figure out what they are. By eliminating duplicate values, we'll either have a single `Bool` value or both `True` and `False`. If both values are present, then we know the symbol isn't pure. If there's only a single value, then we need to know which it is so that we can assign it to the variable.

Now, we'll define the `findUnitClause` function. From the textbook,

> Unit clause heuristic: A unit clause was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model. For example, if the model contains $B = \text{true}$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to false. The unit clause heuristic assigns all such symbols before branching on the remainder.

As above, we'll punt refining the clauses with the model until later.

```haskell
findUnitClause :: Clauses -> Model -> Maybe Assignment
findUnitClause clauses' model = assignSymbol <$> firstUnitClause
  where
    clauses :: Clauses
    clauses = refineUnit clauses' model

    firstUnitClause :: Maybe Literal
    firstUnitClause =
      asum (map (getSingleton .  mapMaybe ifInModel) clauses)

    ifInModel :: Literal -> Maybe Literal
    ifInModel (sign, symbol) =
      case Map.lookup symbol model of
           Just val -> if apply sign val
                       then Nothing
                       else Just (sign, symbol)
           _ -> Just (sign, symbol)

    assignSymbol :: Literal -> Assignment
    assignSymbol (sign, symbol) = symbol := apply sign True
```

This is much simpler than pure symbols! We map assignment over the first unit clause that is found. The first unit clause is found with the `asum` technique. We take the list of clauses, and for each clause, first determine what to do if it's in the model already. If the symbol is in the model, then we check to see if the literal in the clause has a `True` or `False` value by applying the sign to the value. If the value is True, then we don't include it. Otherwise, we include the literal in the list. Finally, we attempt to get the singleton list. `asum` gets the first clause which satisfies these conditions.

Now, in the previous functions, we punted refining the clauses. It's time to do that. For a pure symbol, the given optimization is (from the book):

> Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains $B = \text{false}$, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

We'll start by folding the model and clauses into a new set of clauses. The helper function will go through each symbol in the model, find the relevant clauses, and modify them appropriately.

```haskell
refinePure :: Clauses -> Model -> Clauses
refinePure = Map.foldrWithKey f
  where
    f :: Symbol -> Bool -> Clauses -> Clauses
    f sym val = map discardTrue
      where
        discardTrue =
          filter (not . clauseIsTrue)
```

```
      clauseIsTrue (sign, symbol) =
        symbol == sym && apply sign val
```

The optimization from the text for the unit clause is:

> In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model. For example, if the model contains $B = true$, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must be set to false. The unit clause heuristic assigns all such symbols before branching on the remainder.

```
refineUnit :: Clauses -> Model -> Clauses
refineUnit clauses model = map refine clauses
  where
    refine :: CNF -> CNF
    refine cnf =
      case allButOneFalse cnf of
           Just (s := True)  -> [p s]
           Just (s := False) -> [n s]
           Nothing -> cnf

    allButOneFalse :: CNF -> Maybe Assignment
    allButOneFalse =
      getSingleton . filter (not . getValue) . map assign

    assign :: Literal -> Assignment
    assign (sign, sym) =
      sym := Map.findWithDefault (apply sign True) sym model
```

If all but one of the literals in the CNF are false, then we return that with the proper assigment. Otherwise, we return the whole CNF expression.

Starting from a straight up transcription, we've now finally implemented enough to solve problems!

```
solved :: Maybe Model
solved = dpll ex ["p", "a", "c"] Map.empty
```

The output will be kind of ugly, so let's make a pretty printing function:

```
showModel :: Model -> String
showModel =
  unlines . map (show . snd) . Map.toList . Map.mapWithKey (:=)
```

Evaluating `solved` in GHCi gives us:

```
Prelude HW6> putStr . fromJust $ showModel <$> solved
(a := False)
(c := False)
(p := False)
```

Assigning the variables with those values does return a true model. Nice!

2 Australia

Now, for a less trivial problem – giving Australia a three coloring.

First, we'll want to define our symbols:

```
colors :: [Symbol]
colors = [green, blue, red]

green = "-green"
blue = "-blue"
red = "-red"

states :: [Symbol]
states =
  [ western
  , southern
  , northern
  , queensland
  , newSouthWales
  , victoria
  ]

western  = "Western"
southern = "Southern"
northern = "Northern"
queensland = "Queensland"
newSouthWales = "New South Wales"
victoria = "Victoria"
```

Now we'll express that a given state can have one color, but only one:

```
hasColor :: Symbol -> Clauses
hasColor st =
   [ [ p $ st `is` green
     , p $ st `is` blue
     , p $ st `is` red
     ]
   , [ n $ st `is` blue
```

```
    , n $ st `is` red
    ]
  , [ n $ st `is` green
    , n $ st `is` red
    ]
  , [ n $ st `is` green
    , n $ st `is` blue
    ]
  ]
```

Since our symbols are lists, we can concatenate them together. We don't want to get the two confused, so we make specialized functions that only work on symbols and clauses, respectively.

```
is :: Symbol -> Symbol -> Symbol
is = (++)

(/\) :: Clauses -> Clauses -> Clauses
(/\) = (++)
```

And, since we'll often want to take a list of things, apply a function to each, and make a clause out of the whole thing, we'll alias the `bind` function to something that looks kind of like "take the conjunction of this whole set."

```
(/\:) :: Monad m => m a -> (a -> m b) -> m b
(/\:) = (>>=)
```

At first, we'll simply say that every state has a color.

```
initialConditions :: Clauses
initialConditions = states /\: hasColor
```

Next, we'll say that for a pair of adjacent states, they can't both be the same color.

```
adjNotEqual :: (Symbol, Symbol) -> Clauses
adjNotEqual (a, b) = colors /\: bothAreNot
  where
    bothAreNot color =
      [ [ n $ a `is` color
        , n $ b `is` color
        ]
      ]
```

Next, a list of adjacent states...

```haskell
adjStates :: [(Symbol, Symbol)]
adjStates =
  [ (western, northern)
  , (western, southern)
  , (northern, southern)
  , (northern, queensland)
  , (southern, newSouthWales)
  , (southern, victoria)
  , (southern, queensland)
  , (newSouthWales, queensland)
  , (newSouthWales, victoria)
  ]

adjacentStatesNotEqual :: Clauses
adjacentStatesNotEqual = adjStates /\: adjNotEqual

australiaClauses :: Clauses
australiaClauses =
  initialConditions /\ adjacentStatesNotEqual

australiaSymbols :: Symbols
australiaSymbols =
  is <$> states <*> colors
```

Now, we've finally accomplished the encoding, and we can get the solution.

```haskell
australiaSolution :: Maybe Model
australiaSolution = dpll australiaClauses australiaSymbols mempty
```

It can be printed with the following function:

```haskell
showOnlyTrue :: Model -> String
showOnlyTrue =
  unlines . map (show . snd)
    . filter (getValue . snd)
    . Map.toList . Map.mapWithKey (:=)

printAustralia :: IO ()
printAustralia = do
  let model = fromJust australiaSolution
  putStrLn (showOnlyTrue model)
```

Which, when evaluated in GHCi, gives us:

```
Prelude HW6> printAustralia
(New South Wales-red := True)
```

```
(Northern-red := True)
(Queensland-blue := True)
(Southern-green := True)
(Victoria-blue := True)
(Western-blue := True)
```

This can be verified manually to be a correct coloring of Australia.

3 Resolution

There's a very nice purely functional solution to resolution. In "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire" [Mejier 1991], recursive functions like `map`, `foldr`, and `iterate` are generalized into catamorphisms, anamorphisms, and (my favorite) hylomorphisms. A hylomorphism is a an anamorphism followed by a catamorphism – the unfolding and refolding of a structure.

As it happens, if your types are simple enough, then a hylomorphism is extremely easy to encode:

```
hylo :: (a -> Maybe a) -> a -> a
hylo f a = maybe a (hylo f) (f a)
```

If `f a` evaluates to `Nothing`, then the recursion terminates and we return `a`. Otherwise, we take the `Just` value and recurse.

We can use this to get the fix point of a structure and an expansion function with `hylo (isFixpoint expand)`:

```
isFixpoint :: Eq a => (a -> a) -> a -> Maybe a
isFixpoint f a =
  if a == b
     then Nothing
     else Just b
  where
    b = f a
```

Now, with a suitable `Clauses -> Clauses` function, we can resolve the clauses until they've maxed out.

Unfortunately, my cleverness ran out around this point. Fortunately, despite what you may have heard, Haskell makes for a rather pleasant experience for programming in an imperative style with mutable data.

We'll take advantage of the `ST` monad for safe imperative code.

```
resolution :: Set CNF -> (Bool, Set CNF)
resolution kb' = runST $ do
  new <- newSTRef Set.empty
  go new kb'
```

First, we initialize the empty set variable, and pass the reference into the `go` helper recursive function.

```
  where
    go setRef kb = do
      let clauses = toList kb
          pairs = do
            i <- clauses
            j <- clauses
            guard (i /= j)
            return (i, j)
```

This gets all unequal pairings from the clauses.

```
      h <- for pairs $ \(ci, cj) -> do
        let resolvents = resolve ci cj
        if [] `Set.member` resolvents then
           return True
        else do
           modifySTRef' setRef (Set.union resolvents)
           return False
```

For each pair, we get it's resolvements. If the resolvements contains the empty list, then we can return True and be done. Otherwise, we add the set resolvents to our set. We collect the returned values in the `h` value.

```
      if or h then
          return (True, kb)
      else do
          s <- readSTRef setRef

          if s `Set.isSubsetOf` kb then
              return (False, kb)
          else
              go setRef (kb `Set.union` s)
```

If any value in `h` is true, then we return `True` and the knowledge base. Otherwise, we check to see if the current set is a subset of the knowledge base. If it is, then we return false with the current knowledge base. Otherwise, we recurse, doing another iteration of the loop.

```
resolve :: CNF -> CNF -> Set CNF
resolve as bs = runST $ do
  new <- newSTRef Set.empty
  resolvePosNeg as bs new
  resolvePosNeg bs as new
  readSTRef new
```

We get to do a lot of mutable reference passing in `resolve`. We create a new set, pass it in to both possibilities of `resolvePosNeg`, and then return the created set.

```
resolvePosNeg :: CNF -> CNF -> STRef s (Set CNF) -> ST s (Set CNF)
resolvePosNeg as bs resolvents = do
  let complements = Set.intersection setA setB
      setA = Set.fromList (mkSet Pos as)
      setB = Set.fromList (mkSet Neg bs)
      mkSet s = onlySym . filter ((== s) . fst)
      onlySym = map snd
```

We filter the lists to get the symbols we're looking for. We want the positive symbols from the first list, and the negative symbols from the second list. Then we discard the signs, keeping only the symbols. We take the intersection of these two sets, which gives us the symbols with complementary representations.

```
  for_ complements $ \complement -> do
    resLiterals <- newSTRef Set.empty
    for_ as $ \literal ->
      when (Neg == fst literal || snd literal /= complement) $
        modifySTRef' resLiterals (Set.insert literal)

    for_ bs $ \literal ->
      when (Pos == fst literal || snd literal /= complement) $
        modifySTRef' resLiterals (Set.insert literal)

    newLits <- Set.toList <$> readSTRef resLiterals
    modifySTRef' resolvents (Set.insert newLits)

  readSTRef resolvents
```

For each complementary symbol, this makes a new set of literals, and inserts the appropriate symbols with assignments into the set. Finally it adds all of the new literals into the referenced set and returns that.

```
runProblem :: Clauses -> Clauses -> (Bool, Set CNF)
runProblem kb query =
  resolution (Set.fromList (kb ++ map (map negate) query))
  where
    negate (Pos, a) = (Neg, a)
    negate (Neg, a) = (Pos, a)
```

We can run a problem by providing a knowledge base and the logical sentence we want to prove (in CNF). We want to determine if Western Australia can be green and Victoria is red.

```haskell
ausQuery :: Clauses
ausQuery =
  [ [ p $ western `is` green ]
  , [ n $ western `is` red ]
  , [ n $ western `is` blue ]
  , [ p $ victoria `is` red ]
  , [ n $ victoria `is` green ]
  , [ n $ victoria `is` blue ]
  ]
```

We'll need a way to print the results in a way that makes sense:

```haskell
printClauses :: (Bool, Set CNF) -> IO ()
printClauses =
  putStrLn . unlines . map show . toList . snd
```

Now that we've got all of our print statements in order, we can define `main`.

```haskell
main :: IO ()
main = do
  putStrLn "Solution to Australia coloring problem:"
  printAustralia
  putStrLn "Enumerated clauses via resolution:"
  printClauses (runProblem australiaClauses ausQuery)
```

With the following output, after running `stack runghc hw6.lhs`:

```
ExecOpts {eoCmd = Just ExecRunGhc, eoArgs = ["hw6.lhs"], eoExtra = ExecOptsEmbellished {eoEnvS
Solution to Australia coloring problem:
(New South Wales-red := True)
(Northern-red := True)
(Queensland-blue := True)
(Southern-green := True)
(Victoria-blue := True)
(Western-blue := True)

Enumerated clauses via resolution:
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-green"),(Neg,"New South Wales-blue")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-green"),(Neg,"New South Wales-green")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-green"),(Neg,"Queensland-red")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-green"),(Neg,"Southern-red")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-green"),(Neg,"Victoria-red")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-red"),(Neg,"New South Wales-blue")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-red"),(Neg,"New South Wales-red")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-red"),(Neg,"Queensland-green")]
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-red"),(Neg,"Southern-green")]
```

```
[(Pos,"New South Wales-blue"),(Pos,"New South Wales-red"),(Neg,"Victoria-green")]
[(Pos,"New South Wales-green"),(Pos,"New South Wales-blue"),(Pos,"New South Wales-red")]
[(Pos,"New South Wales-green"),(Pos,"New South Wales-red"),(Neg,"New South Wales-green")]
[(Pos,"New South Wales-green"),(Pos,"New South Wales-red"),(Neg,"New South Wales-red")]
[(Pos,"New South Wales-green"),(Pos,"New South Wales-red"),(Neg,"Queensland-blue")]
[(Pos,"New South Wales-green"),(Pos,"New South Wales-red"),(Neg,"Southern-blue")]
[(Pos,"New South Wales-green"),(Pos,"New South Wales-red"),(Neg,"Victoria-blue")]
[(Pos,"Northern-blue"),(Pos,"Northern-green"),(Neg,"Northern-blue")]
[(Pos,"Northern-blue"),(Pos,"Northern-green"),(Neg,"Northern-green")]
[(Pos,"Northern-blue"),(Pos,"Northern-green"),(Neg,"Queensland-red")]
[(Pos,"Northern-blue"),(Pos,"Northern-green"),(Neg,"Southern-red")]
[(Pos,"Northern-blue"),(Pos,"Northern-green"),(Neg,"Western-red")]
[(Pos,"Northern-blue"),(Pos,"Northern-red"),(Neg,"Northern-blue")]
[(Pos,"Northern-blue"),(Pos,"Northern-red"),(Neg,"Northern-red")]
[(Pos,"Northern-blue"),(Pos,"Northern-red"),(Neg,"Queensland-green")]
[(Pos,"Northern-blue"),(Pos,"Northern-red"),(Neg,"Southern-green")]
[(Pos,"Northern-blue"),(Pos,"Northern-red"),(Neg,"Western-green")]
[(Pos,"Northern-green"),(Pos,"Northern-blue"),(Pos,"Northern-red")]
[(Pos,"Northern-green"),(Pos,"Northern-red"),(Neg,"Northern-green")]
[(Pos,"Northern-green"),(Pos,"Northern-red"),(Neg,"Northern-red")]
[(Pos,"Northern-green"),(Pos,"Northern-red"),(Neg,"Queensland-blue")]
[(Pos,"Northern-green"),(Pos,"Northern-red"),(Neg,"Southern-blue")]
[(Pos,"Northern-green"),(Pos,"Northern-red"),(Neg,"Western-blue")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-green"),(Neg,"New South Wales-red")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-green"),(Neg,"Northern-red")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-green"),(Neg,"Queensland-blue")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-green"),(Neg,"Queensland-green")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-green"),(Neg,"Southern-red")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-red"),(Neg,"New South Wales-green")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-red"),(Neg,"Northern-green")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-red"),(Neg,"Queensland-blue")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-red"),(Neg,"Queensland-red")]
[(Pos,"Queensland-blue"),(Pos,"Queensland-red"),(Neg,"Southern-green")]
[(Pos,"Queensland-green"),(Pos,"Queensland-blue"),(Pos,"Queensland-red")]
[(Pos,"Queensland-green"),(Pos,"Queensland-red"),(Neg,"New South Wales-blue")]
[(Pos,"Queensland-green"),(Pos,"Queensland-red"),(Neg,"Northern-blue")]
[(Pos,"Queensland-green"),(Pos,"Queensland-red"),(Neg,"Queensland-green")]
[(Pos,"Queensland-green"),(Pos,"Queensland-red"),(Neg,"Queensland-red")]
[(Pos,"Queensland-green"),(Pos,"Queensland-red"),(Neg,"Southern-blue")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"New South Wales-red")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"Northern-red")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"Queensland-red")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"Southern-blue")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"Southern-green")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"Victoria-red")]
[(Pos,"Southern-blue"),(Pos,"Southern-green"),(Neg,"Western-red")]
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"New South Wales-green")]
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"Northern-green")]
```

```
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"Queensland-green")]
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"Southern-blue")]
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"Southern-red")]
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"Victoria-green")]
[(Pos,"Southern-blue"),(Pos,"Southern-red"),(Neg,"Western-green")]
[(Pos,"Southern-green"),(Pos,"Southern-blue"),(Pos,"Southern-red")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"New South Wales-blue")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"Northern-blue")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"Queensland-blue")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"Southern-green")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"Southern-red")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"Victoria-blue")]
[(Pos,"Southern-green"),(Pos,"Southern-red"),(Neg,"Western-blue")]
[(Pos,"Victoria-blue")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-green")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-green"),(Neg,"New South Wales-red")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-green"),(Neg,"Southern-red")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-green"),(Neg,"Victoria-blue")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-green"),(Neg,"Victoria-green")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-red"),(Neg,"New South Wales-green")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-red"),(Neg,"Southern-green")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-red"),(Neg,"Victoria-blue")]
[(Pos,"Victoria-blue"),(Pos,"Victoria-red"),(Neg,"Victoria-red")]
[(Pos,"Victoria-green")]
[(Pos,"Victoria-green"),(Pos,"Victoria-blue"),(Pos,"Victoria-red")]
[(Pos,"Victoria-green"),(Pos,"Victoria-red"),(Neg,"New South Wales-blue")]
[(Pos,"Victoria-green"),(Pos,"Victoria-red"),(Neg,"Southern-blue")]
[(Pos,"Victoria-green"),(Pos,"Victoria-red"),(Neg,"Victoria-green")]
[(Pos,"Victoria-green"),(Pos,"Victoria-red"),(Neg,"Victoria-red")]
[(Pos,"Western-blue")]
[(Pos,"Western-blue"),(Pos,"Western-green"),(Neg,"Northern-red")]
[(Pos,"Western-blue"),(Pos,"Western-green"),(Neg,"Southern-red")]
[(Pos,"Western-blue"),(Pos,"Western-green"),(Neg,"Western-blue")]
[(Pos,"Western-blue"),(Pos,"Western-green"),(Neg,"Western-green")]
[(Pos,"Western-blue"),(Pos,"Western-red")]
[(Pos,"Western-blue"),(Pos,"Western-red"),(Neg,"Northern-green")]
[(Pos,"Western-blue"),(Pos,"Western-red"),(Neg,"Southern-green")]
[(Pos,"Western-blue"),(Pos,"Western-red"),(Neg,"Western-blue")]
[(Pos,"Western-blue"),(Pos,"Western-red"),(Neg,"Western-red")]
[(Pos,"Western-green"),(Pos,"Western-blue"),(Pos,"Western-red")]
[(Pos,"Western-green"),(Pos,"Western-red"),(Neg,"Northern-blue")]
[(Pos,"Western-green"),(Pos,"Western-red"),(Neg,"Southern-blue")]
[(Pos,"Western-green"),(Pos,"Western-red"),(Neg,"Western-green")]
[(Pos,"Western-green"),(Pos,"Western-red"),(Neg,"Western-red")]
[(Pos,"Western-red")]
[(Neg,"New South Wales-blue")]
[(Neg,"New South Wales-blue"),(Neg,"New South Wales-red")]
[(Neg,"New South Wales-blue"),(Neg,"Queensland-blue")]
```

```
[(Neg,"New South Wales-blue"),(Neg,"Victoria-blue")]
[(Neg,"New South Wales-green")]
[(Neg,"New South Wales-green"),(Neg,"New South Wales-blue")]
[(Neg,"New South Wales-green"),(Neg,"New South Wales-red")]
[(Neg,"New South Wales-green"),(Neg,"Queensland-green")]
[(Neg,"New South Wales-green"),(Neg,"Victoria-green")]
[(Neg,"New South Wales-red"),(Neg,"Queensland-red")]
[(Neg,"New South Wales-red"),(Neg,"Victoria-red")]
[(Neg,"Northern-blue")]
[(Neg,"Northern-blue"),(Neg,"Northern-red")]
[(Neg,"Northern-blue"),(Neg,"Queensland-blue")]
[(Neg,"Northern-blue"),(Neg,"Southern-blue")]
[(Neg,"Northern-green"),(Neg,"Northern-blue")]
[(Neg,"Northern-green"),(Neg,"Northern-red")]
[(Neg,"Northern-green"),(Neg,"Queensland-green")]
[(Neg,"Northern-green"),(Neg,"Southern-green")]
[(Neg,"Northern-red")]
[(Neg,"Northern-red"),(Neg,"Queensland-red")]
[(Neg,"Northern-red"),(Neg,"Southern-red")]
[(Neg,"Queensland-blue"),(Neg,"Queensland-red")]
[(Neg,"Queensland-green"),(Neg,"Queensland-blue")]
[(Neg,"Queensland-green"),(Neg,"Queensland-red")]
[(Neg,"Southern-blue")]
[(Neg,"Southern-blue"),(Neg,"New South Wales-blue")]
[(Neg,"Southern-blue"),(Neg,"Queensland-blue")]
[(Neg,"Southern-blue"),(Neg,"Southern-red")]
[(Neg,"Southern-blue"),(Neg,"Victoria-blue")]
[(Neg,"Southern-green")]
[(Neg,"Southern-green"),(Neg,"New South Wales-green")]
[(Neg,"Southern-green"),(Neg,"Queensland-green")]
[(Neg,"Southern-green"),(Neg,"Southern-blue")]
[(Neg,"Southern-green"),(Neg,"Southern-red")]
[(Neg,"Southern-green"),(Neg,"Victoria-green")]
[(Neg,"Southern-red")]
[(Neg,"Southern-red"),(Neg,"New South Wales-red")]
[(Neg,"Southern-red"),(Neg,"Queensland-red")]
[(Neg,"Southern-red"),(Neg,"Victoria-red")]
[(Neg,"Victoria-blue")]
[(Neg,"Victoria-blue"),(Neg,"Victoria-red")]
[(Neg,"Victoria-green")]
[(Neg,"Victoria-green"),(Neg,"Victoria-blue")]
[(Neg,"Victoria-green"),(Neg,"Victoria-red")]
[(Neg,"Victoria-red")]
[(Neg,"Western-blue")]
[(Neg,"Western-blue"),(Neg,"Northern-blue")]
[(Neg,"Western-blue"),(Neg,"Southern-blue")]
[(Neg,"Western-blue"),(Neg,"Western-red")]
[(Neg,"Western-green")]
```

```
[(Neg,"Western-green"),(Neg,"Northern-green")]
[(Neg,"Western-green"),(Neg,"Southern-green")]
[(Neg,"Western-green"),(Neg,"Western-blue")]
[(Neg,"Western-green"),(Neg,"Western-red")]
[(Neg,"Western-red")]
[(Neg,"Western-red"),(Neg,"Northern-red")]
[(Neg,"Western-red"),(Neg,"Southern-red")]
```