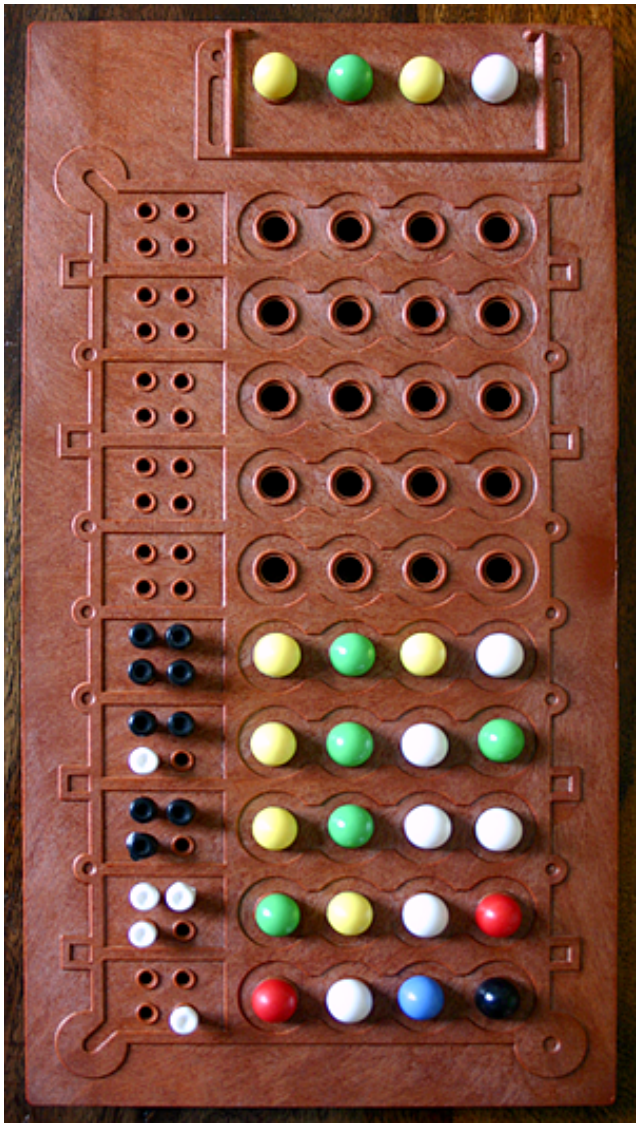
Mastermind

Using SAT methods

The Game

- One player (the master) makes a secret code
- The other player (the mind) attempts to guess that code
- The master gives the mind feedback after each guess



Reasons to use SAT

- A large pool of possible codes is available
- To intelligently guess a code, the mind must use the information from previous guesses as constraints

SBV Library

- SMT Based Verification
- Calls to an external SMT solver to prove properties about Haskell programs
- Introduces symbolic types to be used
    - SBool: Symbolic booleans
    - SWord8: unsigned 8-bit symbolic words
    - etc.

Example

```
sort :: [SWord8] -> [SWord8]
sort []     = []
sort (x:xs) = insert x (sort xs)

insert :: SWord8 -> [SWord8] -> [SWord8]
insert x []     = [x]
insert x (y:ys) = ite (x .<= y) (x:y:ys)
                        (y:insert x ys)
```

Example

```
proveSort4 = prove $ \a b c d ->
                sorted (sort [a,b,c,d])
 where
     sorted [a,b,c,d] =
         a.<=b &&& b.<=c &&& c.<=d
```

Scoring Problem

- Calculating the number of blacks is easy: count how many are pairwise equal between the guess and the code
- How do you calculate the number of whites?

```
score :: [SPeg] -> [SPeg] -> SScore
score xs ys = (b,bw - b)
  where
        b  = sum $ zipWith equal xs ys
        bw = match (sort xs) (sort ys)

match :: [SPeg] -> [SPeg] -> SWord8
match [] _ = 0
match _ [] = 0
match (x:xs) (y:ys) =
   ite (x.==y) (1 + match xs ys)
       (ite (x.<y) (match xs (y:ys))
            (match (x:xs) ys))
```

scoreList

- Checks whether or not a new guess matches the information we have received from prior guesses

```
scoreList :: [SPeg] -> Table -> SBool
scoreList x [] = true
scoreList x ((y,s):ys) = score x y .== s
                         &&& scoreList x ys
```

guess

```
guess :: Table -> IO (Maybe [Peg])
guess xs = do
   res <- sat $ \t -> let g = code2list t in
            scoreList g xs &&& iscode g
   return (extractModel res :: Maybe [Peg])
```

Guessing

- Simple constraints give boring guesses
- Random first guess is better but still gives naive guesses
- predicateGuess allows the use of heuristics

predicateGuess

```
predicateGuess::([SPeg]->SBool)->Table->IO (Maybe [Peg])
predicateGuess p xs = do
    res <- sat $ \t -> let g = code2list t in
              scoreList g xs &&& iscode g &&& p g
    case (extractModel res :: Maybe [Peg]) of
      Just pegs -> return (Just pegs)
      Nothing   -> guess' xs
```

Heuristic: Diversity

- Try to make guesses with unique pegs except for one pair

```
diverse :: [SPeg] -> SBool
diverse x =
  diversity x .== fromIntegral (codeSize - 1)

diversity :: [SPeg] -> SWord8
diversity []     = 0
diversity (x:xs) = diversity xs +
                   ite (x `elt` xs) 0 1
```

Comparison: guess

```
Code of size 6 from [0,1,2,3,4,5,6,7,8,9]
12 guesses
Guess 1 : [0,0,0,0,0,0] : 10
Guess 2 : [1,1,1,1,1,0] : 12
Guess 3 : [2,2,2,0,1,1] : 21
Guess 4 : [8,8,0,9,1,1] : 14
Guess 5 : [9,4,1,0,8,1] : 41
Guess 6 : [3,9,1,0,8,1] : 32
Guess 7 : [9,1,6,0,8,1] : 50
Guess 8 : [9,1,5,0,8,1] : 50
Guess 9 : [9,1,7,0,8,1] : 50
Guess 10 : [9,1,9,0,8,1] : 60
Completed
```

Comparison: diversity

(with random first guess)

```
Code of size 6 from [0,1,2,3,4,5,6,7,8,9]
12 guesses
Guess 1 : [5,4,1,3,3,8] : 12
Guess 2 : [1,3,6,7,8,8] : 02
Guess 3 : [0,1,2,3,4,4] : 04
Guess 4 : [3,0,1,4,9,9] : 13
Guess 5 : [4,4,8,0,1,9] : 50
Guess 6 : [2,4,8,0,1,9] : 60
Completed
```

Further Study

- More heuristics
- How do various heuristics compare in performance?
- evilMaster