Solving the satisfiability problem in Haskell

- Code: https://github.com/elben/sat
- Inspired by Felienne Hermans' Quarto talk at LambdaConf 2016.

Motivation

- Felienne wanted to know if the board game Quarto can end in a tie.
- Today, we'll ask a simpler question: can tic-tac-toe end in a tie?

Imagine if we had...

```
oracle :: Question -> Answer
makeQuestion :: String -> Question

oracle (makeQuestion "Can tic-tac-toe end in a tie?")
-- Yes. Example:
-- XOX
-- OXX
-- OXO
```

How to write the question: “can tic-tac-toe end in a tie?”

» As a boolean proposition.

Boolean satisfiability (SAT)

Is this satisfiable? That is, can you set a and b to make the statement true?

$$(a \land b) \lor \neg b$$

```
(a && b) || !b
```

Boolean satisfiability (SAT)

Is this satisfiable?

$$(a \land \neg b) \lor \neg a$$

Boolean satisfiability (SAT)

Is this satisfiable?

$$(a \land \neg b) \lor \neg a \lor \neg(b \land c \land \neg e) \lor e \lor \neg b \lor (f \land \neg e) \lor \neg b \lor \neg(f \land a \land \neg c) \lor e \lor \neg b$$

- How many rows are in our truth table?
- 2^n

Turns out, SAT is NP-hard

- Running time of $O(2^n)$. $2^{100} = 1267650600228229401496703205376$
- So we're screwed?

SAT solvers to the rescue!

http://minisat.se/

http://www.cs.uni-potsdam.de/clasp/

So what?

- If we can translate our human question to a boolean proposition, then we can find the answer.
- SAT is NP-complete. Any NP-hard problem can be translated to any NP-complete problem.
- Battleship, Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokemon are NP-hard (source 1 source 2)

Input to SAT solvers

input.txt (DIMACS format)

```
p cnf 4 2
1 2 -3 0
4 2 3 0
1 -3 9 -2
…
```

```
$ clasp 3 input.txt

# c clasp version 3.1.3
# c Reading from input.txt
# c Solving...
# c Answer: 1
# v -1 2 -3 4 -5 -6 7 -8 9 10 -11 12 -13 14 15 -16 17 -18 0
# c Answer: 2
# v -1 2 -3 -4 -5 6 7 -8 9 10 -11 12 13 14 -15 -16 17 -18 0
# c Answer: 3
# v -1 2 -3 4 -5 6 7 -8 9 10 -11 12 -13 14 -15 -16 17 -18 0
# s SATISFIABLE
# c
# c Models         : 3+
# c Calls          : 1
# c Time           : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
# c CPU Time       : 0.000s
```

Questions?

SAT solvers use Conjunctive Normal Form (CNF)

Valid:

$$(a \lor b) \land (\neg b \lor c \lor d) \land (d \lor \neg e)$$

Invalid:

$$\neg(b \lor c)$$
$$(a \land b) \lor c$$

Rules to get to CNF

$$\begin{aligned} a &\Rightarrow & a \\ \neg\neg a &\Rightarrow & a \\ \neg(a \wedge b) &\Rightarrow & \neg a \vee \neg b \\ a \wedge (b \wedge c) &\Rightarrow & a \wedge b \wedge c \\ a \vee (b \wedge c) &\Rightarrow & (a \vee b) \wedge (a \vee c) \end{aligned}$$

» Identity, double negation, DeMorgan’s, associative, distributive.

Looks like we’re writing a compiler!

Haskell is awesome for writing compilers

» Best-of-class parsers. Parsec and Attoparsec.

» Abstract syntax tree maps really well to algebraic data types.

» Safe and fun for the whole family (statically typed, immutable).

» Pugs for Perl 6 was written in Haskell. Elm.

Overview

1. Convert the question “can tic-tac-toe end in a tie?” to a boolean expression.
2. Normalize expression to CNF.
3. Convert this boolean statement to the SAT solver format.
4. Run it through the solver.
5. Two possible answers: YES and an example, or NO.

Algebraic data types (sum types)

```haskell
data Bool = True | False

-- >>> :t True
-- True :: Bool
```

Algebraic data types (sum types)

```
data Tree a = Node (Tree a) a (Tree a)
            | Nil


--   100
--      \
--       200
(Node Nil 100 (Node Nil 200 Nil)) :: Tree Int
```

Abstract syntax tree

```
data Term = Var String
          | Not Term
          | Or  [Term]
          | And [Term]
```

Abstract syntax tree

```
data Term = Var String
          | Not Term
          | Or  [Term]
          | And [Term]
```

$$(a \land \neg b) \lor \neg a$$

```
Or [And [Var "a", Not (Var "b")], Not (Var "a")]
```

```
-- Parser
parse "a && !!b"
And [Var "a", Not (Not (Var "b"))]

-- Convert to CNF (optimizer)
cnf (And [Var "a", Not (Not (Var "b"))])
And [Var "a", Var "b"]

-- Convert to DIMACS format (compiler)
emit (And [Var "a", Var "b"])
-- p cnf 2 1
-- 0 1
```

Questions?

Interpreter: Convert to CNF

```
cnf :: Term -> Term
cnf (Var n) = Var n
```

```
--        !a  ==> !a
--     !(!a) ==> a
-- !(a v b) ==> !a ^ !b   (De Morgan's)
-- !(a ^ b) ==> !a v !b   (De Morgan's)
cnf (Not term) =
  case term of
    Var n     -> Not (Var n)
    Not t     -> cnf t
    Or  terms -> cnf (And (map Not terms))
    And terms -> cnf (Or (map Not terms))
```

```haskell
-- Associative property: a ^ b ^ (c ^ d) => a ^ b ^ c ^ d
flatten :: Term -> Term

-- Erase useless terms: And [Var "a"] => Var "a"
erase :: Term -> Term

-- a ^ (b ^ c) => (a ^ b ^ c)
-- a ^ (b v c) => a ^ (b v c)
cnf (And terms) =
  let terms' = map cnf terms
  in erase (flatten (And terms'))
```

The `Or` case is the only tricky one.

```
cnf :: Term -> Term
cnf (Or terms) =
  let terms' = map cnf terms
  in if isCnf (Or terms')
     then erase $ flatten $ Or terms'
     else
       let terms'' = distributeOnce terms'
       in cnf $ erase $ flatten (Or terms'')
```

Remember, CNF looks like:

$$a \wedge (b \vee c) \wedge d$$

So what about this:

$$a \vee (b \wedge c) \vee (d \wedge e)$$

- $a \lor (b \land c) \lor d$
- $\Rightarrow ((a \lor b) \land (a \lor c)) \lor d$
- $\Rightarrow (a \lor b \lor d) \land (a \lor c \lor d)$
- `Or` case has to iterate each pair and run distributive law

Demo `cnf` in console

- Tested with QuickCheck (specification testing).

Emit DIMACS format

Keep a mapping of variables to integer representation.

```
emit (And [Var "a", Var "b"])
-- p cnf 2 1
-- 0 1
```

```haskell
emit :: Term -> StateT Env (State Counter) String
emit (Var n) = do
  env <- get
  let m = M.lookup n env
  s <- case m of
         Just i ->
           return i
         Nothing -> do
           i <- lift fresh
           lift (put i)
           modify (M.insert n i)
           return i
  return $ show s
emit (Not t) = do
  ts <- emit t
  return $ "-" ++ ts
emit (Or terms) = do
  s <- mapM emit terms
  return $ unwords s
emit (And terms) = do
  s <- mapM emit terms
  return $ intercalate " 0\n" s ++ " 0"
```

Demo `emit`

```
putStrLn (emitDimacsWithDebug True (And [Var "a", Var "b"]))
```

Questions?

Back to tic-tac-toe

Tic Tac Toe

Can it end in a tie?

XOX

OXX

OXO

How do you specify, using only boolean propositions, that the game has ended in a tie?

```
123
456
789

x1 = player "X" is in position 1
o2 = player "O" is in position 2
```

Player X is not in a winning position.
Player O is not in a winning position.

$$\neg(x_1 \land x_2 \land x_3) \land$$
$$\neg(o_1 \land o_2 \land o_3) \land$$

And so on...

123	XXX	OOO
456	???	???
789	???	???

A position *must* be played by either X or O.

$$(x_1 \lor o_1) \land (x_2 \lor o_2) \land \ldots$$

A position can only be played by *either* X or O.

$$\neg(x_1 \land o_1) \land \neg(x_2 \land o_2) \land \ldots$$

The game is played out. That is, X gets 5 moves and O gets 4 moves.

???

The game is played out. That is, X gets 5 moves and O gets 4 moves.

This is true by the constructs we made.

We can write a proof to show that if X has *more* than 5 plays, then X *must* be in a winning position.

```haskell
-- TicTacToe.hs

winningPositions :: [[Int]]
winningPositions =
  [[(r*3)+1..(r*3)+3] | r <- [0..2]] ++
  [[c,c+3,c+6] | c <- [1..3]] ++
  [[1,5,9], [3,5,7]]

-- Converts ["a", "b", "c"] to (!a v !b v !c)
negateOrs :: [String] -> Term
negateOrs terms = Or (map (Not . Var) terms)

-- Statement that player is not in a winning position.
playerNotInWinningPosition :: String -> Term
playerNotInWinningPosition player =
  let positions = map (map (\cell -> player ++ show cell)) winningPositions
  in And (map negateOrs positions)

-- Every location has one of two players on it.
everyLocationFilled :: Term
everyLocationFilled = And [Or [Var ("x" ++ show c), Var ("o" ++ show c)] | c <- [1..9]]

-- Every location occupied only by one player.
playerNotInSamePosition :: Term
playerNotInSamePosition = And [Or [Not (Var ("x" ++ show c)), Not (Var ("o" ++ show c))] | c <- [1..9]]

canEndInTie :: Term
canEndInTie =
  let And p1Terms = playerNotInWinningPosition "x"
      And p2Terms = playerNotInWinningPosition "o"
      And x       = playerNotInSamePosition
      And y       = everyLocationFilled
  in And (p1Terms ++ p2Terms ++ x ++ y)
```

```haskell
-- Every location has one of two players on it.
everyLocationFilled :: Term
everyLocationFilled =
  And [Or [Var ("x" ++ show c), Var ("o" ++ show c)]
      | c <- [1..9]]
```

$$(x_1 \vee o_1) \wedge (x_2 \vee o_2) \wedge \ldots$$

```
cd ~/code/sat
stack ghci
import TicTacToe
canEndInTie

putStrLn (emitDimacs canEndInTie)

Ctrl-D

clasp 3 tictactoe.txt
```

Thanks!

Extras

Clique in a graph

```
    3----4
   / \   |
  2---5--6
 /
1
```

```haskell
-- :l Clique
putStrLn $ emitDimacsWithDebug True $ buildGraph 6 3 [(1,2), (2,3), (2,5), (3,4), (3,5), (4,6), (5,6)]
```

Clique rules

$y_{i,r}$ is true if node i is in position r in the clique.

1. There is a node in every clique position.
2. A node cannot occupy multiple clique positions.
3. If there is no edge between two nodes, then those two nodes cannot be in clique.

http://blog.computationalcomplexity.org/2006/12/reductions-to-sat.html

Attribution

- Oracle of Delphi - http://chrisappel.deviantart.com/art/Oracle-of-Delphi-201288884