Assignment #3

Posted on March 11
Due by March 24, 5:30pm

Question 1: Programming the Invincible Tic-Tac-Toe Player

Before you tackle the World Isola Championships, let us consider a simpler game: Tic-Tac-Toe. The complete code for playing a game human vs. computer can be found in the file TicTacToe.hs (or TicTacToe.fr if you prefer) on the course homepage. Well, the code runs but the computer player is not very smart, to say the very least. It simply puts its next piece into the first (in reading direction) open square on the board.

(a) Your first task is to make the program smarter. Implement the minimax algorithm to improve the findBestMove function. This function receives the current state of the game and the number of moves to look ahead. It returns the move, i.e., an integer between 0 and 8, that the computer should play. You do not have to implement alpha-beta pruning.

(b) Bonus Question: Once the computer player is playing better, further improve its play by coming up with a more powerful evaluation function. Ideally, your program should be invincible if it looks only 5 moves ahead.

Question 2: Algorithmic Inference – Sledgehammer Style (required for CS670, bonus question for CS470)

In order to test whether a logical argument is valid, i.e., whether its conclusion follows from its hypotheses, we can use a set of inference rules, resolution refutation, or many other reasoning principles. However, with the help of a computer, we can also simply use a brute-force method.

All we really need to show is that whenever all hypotheses are true, then the conclusion is also true. If you show that, then you have proven the argument to be valid. If you find at least one case in which all hypotheses are true but the conclusion is not, then you discovered a counterexample, and you have shown the argument not to be valid. In order to do this, we can simply test all possible cases. The only things that can vary are the truth values of the atoms, or let us call them “variables” because that’s a term we are more used to. Each of our variables P, Q, etc. can either be true or false. This means that
for all variables that occur in our argument, we have to test all possible interpretations, i.e., assignments of true/false values to all of them, until we exhaust all interpretations or find a counterexample. Implemented on a computer, this algorithm should be able to handle all small-scale problems; even if we have 10 different variables, the $2^{10} = 1024$ cases that need to be tested should be no problem at all.

In order to deal with expressions in predicate logic in Haskell, I wrote some code in a file named “logic.hs” that is available from the course homepage. First, we need a suitable data type. Let us say that all our variables are represented by single capital letters, and we call our connectives IMP, OR, AND, and NOT. Also, we need the values TRUE and FALSE. As we discussed before, the connectives can bind one or two expressions (wffs) to form another, single expression. As a consequence, logical expressions have a hierarchical structure that can best be represented by trees. Given the increasing precedence along IMP, OR, AND, and NOT, the expression

$$P \land Q \land R \lor \lnot S$$

has the following tree representation:

```
IMP
  /
 AND  OR
   /    /
  P    Q  R
     /
    NOT
     /
      S
```

In the tree representation, it is now unambiguous how to evaluate such an expression; once we know the truth values of all variables, we can start from the leaves and apply the operators to their children (the leaves), and then replace the operators with the result. At the end, only one value remains, which is the truth value of the entire initial expression.

In Haskell, we define this data type like this:

```haskell
data Exp = TRUE | FALSE
  | VAR Char
  | NOT Exp
  | AND Exp Exp
  | OR Exp Exp
  | IMP Exp Exp
deriving (Eq)
```
The example expression above could be assigned to a value \( e \) like this:

```haskell
*Main> let e = IMP (AND (VAR 'P') (VAR 'Q')) (OR (VAR 'R') (NOT (VAR 'S')))
```

Thanks to a specification of printing data of type \( \text{Exp} \) (also in logic.hs), we get to see these expressions in infix format, which I think is better readable for humans:

```haskell
*Main> e
((P AND Q) IMP (R OR (NOT S)))
```

Note that these outputs include parentheses for all subexpressions, regardless of operator precedence. This way it is better for us to see what the actual tree structure is. Also, it is much easier to program. Of course, entering expressions as in the “let” assignment above is rather tedious. It would be much nicer if we could simply enter expressions in the form “\( P \ AND \ Q \ IMP \ R \ OR \ NOT \ S \)”, which would then be translated into a tree while accounting for operator precedence. This can be done but includes some difficulties that are beyond the topics of this AI course. If you have taken CS420 or CS451, then you already have a good idea of what needs to be done; it has to do with context-free grammars. I uploaded to the course homepage some text I wrote about 100 years ago, which deals with a very similar problem, namely evaluating arithmetic expressions. You can take a look at it if you like, but you do not have to understand this for the present course, I will never ask you anything about grammars etc. But for the sake of completeness, our grammar for logical expressions is:

```
E -> E IMP D | D
D -> D OR C | C
C -> C AND L | L
L -> NOT A | A
A -> (E) | VAR | TRUE | FALSE
```

One consequence of using this grammar is that we cannot write “NOT NOT Q” but have to write “NOT (NOT Q)”; this only applies to double negation, I hope you can live with that.

There are several functions in logic.hs enabling this parsing of expressions, but all you need to do is apply the “parse” function. Let us try it out:

```haskell
*Main> parse "P AND Q IMP R OR NOT S"
((P AND Q) IMP (R OR (NOT S)))
```

It works! If we now want to evaluate such expressions, we need a function that takes an expression and also the truth values of all variables involved. You specify the truth values
in a string by either just listing the variable name (then the variable is TRUE) or “NOT” followed by the variable name (then the variable is FALSE). Then the “vars” function parses your string into a list of associated variable names and values. For example:

```*Main> vars "P, Q, NOT R, NOT S"
[(P,True),(Q,True),(R,False),(S,False)]```

Let us say value e is our expression and value v is our variables (or interpretation), then we can evaluate the expression under this interpretation using the “eval” function:

```*Main> eval e v
True
```

You now have all the tools you need to build your sledgehammer style inference algorithm. If your hypothesis is given by a string h (multiple hypotheses can be combined into one using AND connectives) and your conclusion is described by a string c, then your function

```
infer h c```

will output either “valid” or “not valid for…” For example:

```*Main> infer “P AND Q” “P”
valid
```

```*Main> infer “P OR Q OR S” “P AND Q”
not valid for P, NOT Q, NOT S```

**Question 3: Resolving Problems – Paper and Pencil Style**

Are the following arguments valid or not? Use resolution or resolution refutation to find out. Proceed in four steps:
- Extract the propositions from the argument and name them with single letters.
- Describe the hypotheses and the conclusion in propositional calculus.
- Convert these expressions into conjunctive normal form (CNF) suitable for resolution or resolution refutation (whichever method you decide to use).
- List the steps of the resolution (refutation) and state the result, that is, whether the argument is valid or not.

If you wrote the “infer” function, use it to verify your result (and the function).
a) If Kevin Flynn loses his disc or his AI system (CLU) cannot be turned off, Kevin Flynn is trapped on the grid. If CLU has no “off” switch, it cannot be turned off. Kevin Flynn does not lose his disk. CLU has no “off” switch. Therefore, Kevin Flynn is trapped on the grid.

b) Anne goes to the library on Saturday or on Sunday. John also goes to the library on Saturday or Sunday. Therefore, both Anne and John will be at the library on Saturday, or both of them will be at the library on Sunday.

c) If the AI lecture is boring, I will fall asleep. If I fall asleep, I will hit my head on the desk. If I hit my head on the desk, I will go to the hospital. I did not go to the hospital. Therefore, the AI lecture is not boring.

d) Jennifer is either watching a soccer game or a baseball game, or both. Whenever she watches a soccer game, she wears a Beckham t-shirt. Whenever she watches a baseball game, she wears a Red Sox scarf. Therefore, Jennifer is wearing a Beckham t-shirt and a Red Sox scarf.

**Question 4: Algorithmic Inference – Refutation Style**

It should also be possible to implement the resolution refutation algorithm in Haskell without too much effort. The greatest problem is getting the logical expression into CNF. To make your life easier, I already came up with functions to remove all IMPs, limit the score of the NOTs, turn everything into CNF (i.e., prevent any AND from being a child of an OR), make a list of clauses, and then eliminate any double-listed variables in each clause. The function textToCNF applies all of these functions in sequence:

```
textToCNF xs = map nub $ listCNF $ makeCNF $ limitNOT $ removeIMP $ parse xs
```

If we consider the example with Gary, once we added the negated conclusion for resolution refutation, we can get the whole thing into CNF using this function:

```
*Main> textToCNF "(I OR A) AND (I IMP C) AND NOT C AND NOT A"
[[I,A],[((NOT I),C),[(NOT C)],[((NOT A)]]
```

Your task is to write a function “inferRR” that gets the same input as “infer” from Question 2, uses resolution refutation and either outputs “valid” or “not valid.”

You can test your function with the arguments in Question 3.