CMPT 383: Assignment #2

Anders Miltner
miltner@cs.sfu.ca

Due Oct 4

Introduction

In this assignment, we will be using Haskell to implement a program synthesizer for boolean expressions.

Boolean expressions are expressions over a set of variables that evaluate to True or False. The variables themselves can only take the values of True or False. The expressions consist of:

- Base Values: True or False
- Variables: Strings that take the values of True or False
- Ands: A conjunction of two other boolean expressions – to evaluate to True both of the subexpressions must evaluate to True
- Ors: A disjunction of two other boolean expressions – to evaluate to True one of the subexpressions must evaluate to True
- Not: A negation of a boolean expression – to evaluate to True the subexpression must evaluate to False

These expressions are formalized in Haskell as the `Expression` data type in Language.hs. The way these expressions evaluate is formalized in Haskell as `evaluate` in Language.hs. To evaluate one requires an assignment of booleans to the variables in the expression. Given an assignment $v_1 \mapsto b_1, \ldots, v_n \mapsto b_n$, a boolean expression e evaluates to a boolean value b is written $e[v_1 \mapsto b_1, \ldots, v_n \mapsto b_n] = b$. Another shorthand for this is $e[\overline{v} \mapsto \overline{b}] = b$ where $\overline{v} = v_1 \ldots, v_n$ and $\overline{b} = b_1 \ldots b_n$.

A program synthesizer is a tool that takes a specification of some form, and outputs an expression that satisfies the specification. In this assignment, we will be building a program synthesizer that generates boolean expressions over a set of variables $\overline{v}$. The specification for this comes as a list of pairs of an assignment of booleans to the variables $\overline{v_i} \mapsto \overline{b_i}$ and an output boolean b_i. Given a list of assignments and outputs as specification $(\overline{v}_1 \mapsto \overline{b}_1, b_1), \ldots, (\overline{v}_n \mapsto \overline{b}_n, b_n)$ an expression satisfies the specification if $\forall i. e[\overline{v_i} \mapsto \overline{b_i}] = b_i$.

The way this synthesizer works is by building iteratively larger expressions. It first builds up expressions of size 1, and sees if any expression satisfies the specification. Next it builds up expressions of size 2, and sees if any expression satisfies the specification. It continues doing this until it either finds a satisfying expression, or it does not find any up to some provided max size. The bulk of the complexity of this assignment lies in correctly generating expressions of a given size.

1 Warm-Up: `numberSplit`

Before we start generating any expressions, we need to define a helper function. The `numberSplit` function splits a positive number into a list of the pairs of positive numbers that sum to it. So, the number 5 has 4 ways to split it: $(1, 4), (2, 3), (3, 2), (4, 1)$. Generally, the number n will have $n - 1$ ways to split it. The order these pairs are returned in does not matter.

2 Warm-Up: `baseExpressionsAtSize` and `varExpressionsAtSize`

Next we want to build synthesizers for base expressions and variable expressions. If you look at `expressionSize` in Language.hs, you can see that the size of a base expression and a variable expression is always 1. So, these functions always should return an empty list when the provided size is not 1. When the provided size is 1, the function should return every base expression (for `baseExpressionsAtSize`) and every variable expression (for `varExpressionsAtSize`).

3 `notExpressionsAtSize`

The function `notExpressionsAtSize` is a little more complex. Note that it takes in a function of type `Int -> [Expression]`. This function describes how to create expressions of a given size.

This is an example of a *dependency injection*. Namely, rather than requiring `notExpressionsAtSize` to recursively call the full function `expressionsAtSize`, we leave this recursive call parameterized. This enables two things. First, the code is more modular. The function for `notExpressionsAtSize` is not tied to `expressionsAtSize`. Rather, the function exists independently, and changes to `expressionsAtSize` does not impact or alter the `notExpressionsAtSize` code. Second, the code is more testable. Because the code for `notExpressionsAtSize` is not tied to `expressionsAtSize`, bugs in it, or any of the code it relies on (like `andExpressionsAtSize` and `varExpressionsAtSize`) does not make the tests for `notExpressionsAtSize` fail.

The resulting values for `notExpressionsAtSize` should all be "not expressions." In otherwords, the outermost constructor should be `ENot`. Note that you want to return all not expressions of a given size k. The size of a not expression is 1 plus the size of the subexpression.

4 `andExpressionsAtSize` and `orExpressionsAtSize`

For the `andExpressionsAtSize` and `orExpressionsAtSize`, you want to follow a similar pattern to `notExpressionsAtSize` However, there's an added complexity. For `notExpressionsAtSize`, you had to simply call the provided input function once on a single number. Now, you must look at all pairs of numbers that sum to the desired size k. This will require the helper function `numberSplit`.

However, even with `numberSplit`, this function is more complex. There are a large number of possible splits – and for each split there are a potentially large number of expressions generated for the left and right subexpression. How do we combine these? A simple `fmap` will not be sufficient.

Fortunately, we can use the list monad. The list monad enables programmers to easily combine lists, and lists of lists. To get full credit for this portion of hte assignment, you *must* use the list monad and do notation to implement `andExpressionsAtSize` and `orExpressionsAtSize`.

5 `expressionsAtSize`

Now you need to combine all these functions together. The `expressionsAtSize` function must generate expressions of every form at a given size, then concatenate them all. The hardest part is finding out what arguments to provide to `notExpressionsAtSize`, `andExpressionsAtSize` and `orExpressionsAtSize`. Namely, what function should be provided to them, such that when those functions call it, they generate expressions of every form at a given size?

6 `expressionSatisfiesExamples`

Next, you need to figure out if your expressions satisfy a given set of examples. An expression satisfies a given set of examples if they all evaluate to the desired output when provided the provided boolean assignment to variables.

7 generator

Almost done! Now you just have to put it all together. At this point, you want to generate expressions of iteratively larger size. If no expressions at a given size are found, you should move onto the next size. This continues until an expression of the given size is found, or until the maximum size provided is passed. Note that you *must* search through expressions of increasing size. If there is an expression of size 3, and you return an expression of size 5 that satisfies the specification, that expression is considered incorrect.