Project 4: expression simplification using synthesis

October 13, 2017

1 Project description

The goal of this project is to build a program to simplify boolean and linear integer arithmetic expressions. To achieve this, you will use a program synthesis tool called Rosette and a your custom partial evaluation procedure.
The problem is the following: given an expression (defined by the grammar in section 3), can you find an equivalent expression such that the size of the expression is smaller?
In order to do so, we ask you to use a program synthesis tool, Rosette. The main difficulty with synthesis is to design the search space for the programs to synthesize. For this project, you will need to design this search space, with two components: a grammar for the synthesizable expressions, and a skeleton to be completed by expressions in this grammar.
But it is also sometimes useful to add your own simplification procedures. For example, you know that `(+ a 0) = a` so if you encounter the expression on the left hand side of the equality you can rewrite it as just `a`. You should use both in conjunction in your expression simplifier.

Deliverables At the end of the project, you will be asked to provide the **source code** of your tool. It should be well commented and easy to understand. We also should be able to test your tool on our inputs, so it should be compilable on the Teaching Lab's machines.
Your tool should either be a library with an API, or an executable than can read files where expressions are written. In both cases, it should not be difficult to test it!

Additionally, we would like to have **a small report** (2-3 pages), explaining:

- the simplification rules your tool uses internally.
- the expression grammar your designed for the synthesis. You don't need to explain the whole grammar, but rather justify your design choices.
- how-to use your tool. If it is a library, a documentation will be needed.

Notation To mark this project we will look at the capacity of your tool to handle various expressions. You will be provided with a set of examples, and you will be evaluated on the ability of your tool to synthesize simplified expression in a limited amount of time. You will be graded propotionally to the number of tests passed over 60 points. The quality of the synthesizable grammar will be graded on 20 points, and 20 points for the expressiveness of the simplification rules of your tool.

Bonus

- You can easily try to extend the input language of your tool. However, be careful when adding non-linear operators!
- You designed some simplification procedure of your own in this tool. You can also use Z3 instead, using its built-in theories. But then you have to be careful about the expression complexity.

2 Language

For this project, since you use Rosette, you are required to use Racket at least partially. You can also plan to program your tool in another language and use Rosette 'under the hood'.

3 Expression grammar

In this project, you will use a simple grammar for expressions, and the basic types of Rosette (Boolean, Integers) and bitvectors. If you want to go further, you can start adding real numbers.

⟨*expression*⟩ ::= ⟨*identifier*⟩
  | ⟨*constant*⟩
  | '(' ⟨*expression*⟩ ')'
  | ⟨*expression*⟩ ⟨*binary-operator*⟩ ⟨*expression*⟩
  | ⟨*unary-operator*⟩ ⟨*expression*⟩
  | ⟨*expression*⟩ '[' ⟨*expression*⟩ ']' // Array access
  | ⟨*expression*⟩ ? ⟨*expression*⟩ : ⟨*expression*⟩ // Conditional expression

⟨*binary-operator*⟩ ::= '+' | '-' | 'min' | 'max' // Arithmetic
  | '&&' | '||' // Boolean
  | '&' | '|' | '^ // Bitwise
  | '==' '>' | '<' | '>=' | '<=' // Comparisons

⟨*unary-operator*⟩ ::= '-' // Arithmetic
  | '!' // Boolean not
  | ' ' // Bitwise not

References